Phuong Nguyen

Procedural Virtual World Generation in a Massively Multiplayer Online Game

Helsinki Metropolia University of Applied Sciences

Bachelor's Degree

Information Technology

Bachelor's Thesis

25 April 2016

Helsinki Metropolia University of Applied Sciences

Authors	Phuong Nguyen
Title	Procedural Virtual World Generation in a Massively Multiplayer Online Game
Number of Pages	50 pages
Date	25 April 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Instructors	Antti Laiho, Senior Lecturer Rashad Hasanzade, Game Designer Toni Nylund, Game Designer Ville Sillanpää, Backend Programmer

The purpose of this thesis was to create a virtual world platform in a massively online multiplayer game through the usage of procedural generation. The target for the practice is the Galaxy Map implementation in Last Planets, a massively multiplayer mobile game aiming at revolutionizing social gameplay, developed by Vulpine Games. The study introduces the business challenge that Vulpine Games was facing, requiring a newly designed and robust replacement for the existing placeholder implementation of the Galaxy Map.

The research approach in this study is action research, which involves the analysis of the existing solution to look for problems. This leads to the literature research process, where good solutions can be found to apply to the current context. The changes to the implementation are then evaluated against the original requirements, which ultimately yield suggestions for future improvement iterations.

The study focuses on how to improve the existing Galaxy Map through the process of using various procedural generation techniques and algorithms to create a persistent and unique environment for a multiplayer game using Unity and C# on the client side. The study also covers the technical solutions when dealing with a mass number of objects in a video game as well as how to manage and optimize network resource usage in such a multiplayer environment.

Based on the results of the final implementation, the project succeeded in creating a virtual world platform which is entirely procedurally generated through the usage of various solutions discovered during the study. Despite rare occasional crashes that happen on low-end devices, the implemented Galaxy Map has managed to meet the requirements presented by Vulpine Games' designers and achieved a reasonable network utilization performance. The knowledge explored and used for the implementation is still rather basic. Procedural generation in video game is a very broad topic with many challenging difficulties from both technical and design perspectives. This suggests that the Galaxy Map can be improved further in the future by using more advanced techniques, especially when more demanding requirements surface. This study, however, can be considered a good fundamental reference for developers who aim to achieve similar virtual world design by using procedural generation.

Key words	game development, social game, mobile game, procedural generation, Unity, C#

Contents

1 Introduction

The video game industry has gone a long way during the last few decades. While the early days of the video game industry started with most focus being on single-player video game, the internet and mobile revolution have been changing the way we play at a dazzling pace. Multiplayer gaming is now being heavily focused on with the core being the interaction between players. Creation of a virtual world, where players can interact and socialize is not feasible and require thoughtful design, and at the same time, the process has to tackle many technical challenges.

1.1 Case Company Background

The case company in this study is Vulpine Games, an independent video game company who put emphasis on social network game. With the current active game Last Planets being under heavy development, the game aims to bring real-time online strategy gaming to the market. Being the first major title of the studio, Last Planets strives to achieve a large scale multiplayer experience on mobile device. Each player will be provided with his/her own planet to develop and compete with other players. The interaction between players is virtualized through a virtual world called the Galaxy Map where the players can find all other players and form either a competitive or collaborative relationship by using the alliance system.

1.2 Business Challenge

The Galaxy Map is already implemented with the basic functionalities. However, the implementation is a placeholder to allow other in-game features to be developed at the same time. The current system creates the entire galaxy and all objects in it right at the beginning. This puts a lot of stress on consumer devices, especially low-end ones, as it causes a serious performance problem, which disallows the support for those devices, thus shrinking the target consumer market. Moreover, all data of the map will be fetched from the server. This creates stress on limited server processing power and limited bandwidth capacity, thus risking gameplay experience for all players connected to the same server. This is severely signified by the fact that the game is a massively online multiplayer game where thousands of players can connect at the same time.

The drawback does not end there as the current map is not capable of being expandable to accommodate more players as the user base increase. Since there is a shortage in human resources, the map cannot be expanded by designing and placing objects by hand, but needs to be generated algorithmically while still having support for more varied, customizable objects on the map along with a unique recognizable pattern across the universe which encourage exploration. Without a good deterministic generation algorithm, the map will be limited in terms of functionality and appearance, preventing it from achieving the design goals.

In order to advance the development of Last Planets towards the goal of having a virtual platform to facilitate the interaction amongst players, the Galaxy Map needs to go through a major technical evaluation, redesign and reimplementation to match the game designer's requirements.

1.3 Objective

The objective of this thesis is to study how to generate a virtual world system computationally for a massively multiplayer online game, specifically the Galaxy Map for Last Planets at Vulpine Games. This method, as opposed to manual generation, is called procedural generation. The study includes research from both technical design and implementation perspectives to tackle the most common obstacles while implementing such a system, surrounded by requirements that need to be met for the game.

The documentation in this study will focus on the most challenging aspects during the implementation process instead of the detail of every single feature that exists in the Galaxy Map system. Any details regarding graphical user interface elements will be omitted, as well as any modules that do not serve the procedural generation system. This is due to the limited scope and size of the study, thus allowing more emphasis on major issues during the implementation of a procedural generation system. These issues will most likely be encountered by any games with similar virtual world design goals. This will make the study a good reference for similar projects, serving as an example of how such a system should be done.

1.4 Structure of the Study

The report is structured with an introduction followed by a section detailing the study approach and design, explaining how the study is conducted to achieve the objective and outcome. Then a whole section on the current state analysis of the Galaxy Map implementation will denote the strengths and weaknesses of the system, along with the design requirements to outline the biggest issues that need to be addressed. Chapter 4 will then document the research process through the work of finding the most necessary information and good practices to conclude how the Galaxy Map in Last Planets should be implemented. The implementation details will follow in chapter 5, where the discussion of major problems and how they are solved specifically for the Galaxy Map will receive an in-depth explanation. Finally in chapter 6 and 7, the result of the study and work will be evaluated and concluded with a mention of future possibilities for improvement.

2 Study Method

This section explains what kind of study approach was used in this report. Then it will proceed to describe the step-by-step flow of how the study was carried out.

2.1 Research Approach

First we consider the nature of this study which is based on a fixed context, bounded within a situation where an organization needs to improve an existing solution by solving certain problems. To improve the existing solution, it should be first diagnosed for its strengths and weaknesses. Since there is a need to improve it according to the requirements, we can identify the problems. Once done, a data collection process of contemporary literature can be carried out to look for similar situations where alternative courses of action can be used to solve the identified problems. The action is then taken to address the issues, before the solution is evaluated again to draw out conclusions and findings.

The best method that describes the process above is action research, which is typically cyclical and consists of five phases: diagnosing, action planning, action taking, evaluating, and specifying learning. (Susman et al. 1978, 588.)

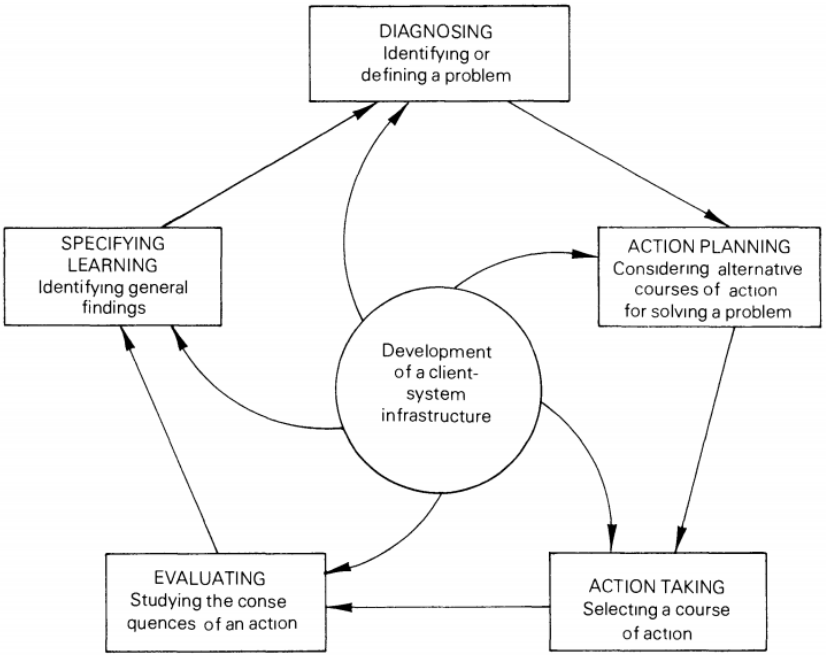


Figure 1. The cyclical process of action research
(Copied from Susman et al. 1978, 588)

Given the cyclical nature of action research method in Figure 1, the study in this report is only one cycle amongst the many cycles of improvement for the Galaxy Map implementation. The solution will always be evaluated in the end, giving room for potential improvements the next time the cycle begins.

2.2 Research Design

The research is divided into five different stages, according to the typical action research cycle. These are steps that will be done in order to implement the Galaxy Map system, beginning with some preliminary problem identification, then the solution researching phase before going through the design and implementation process which will be ultimately evaluated and concluded for further future iteration.

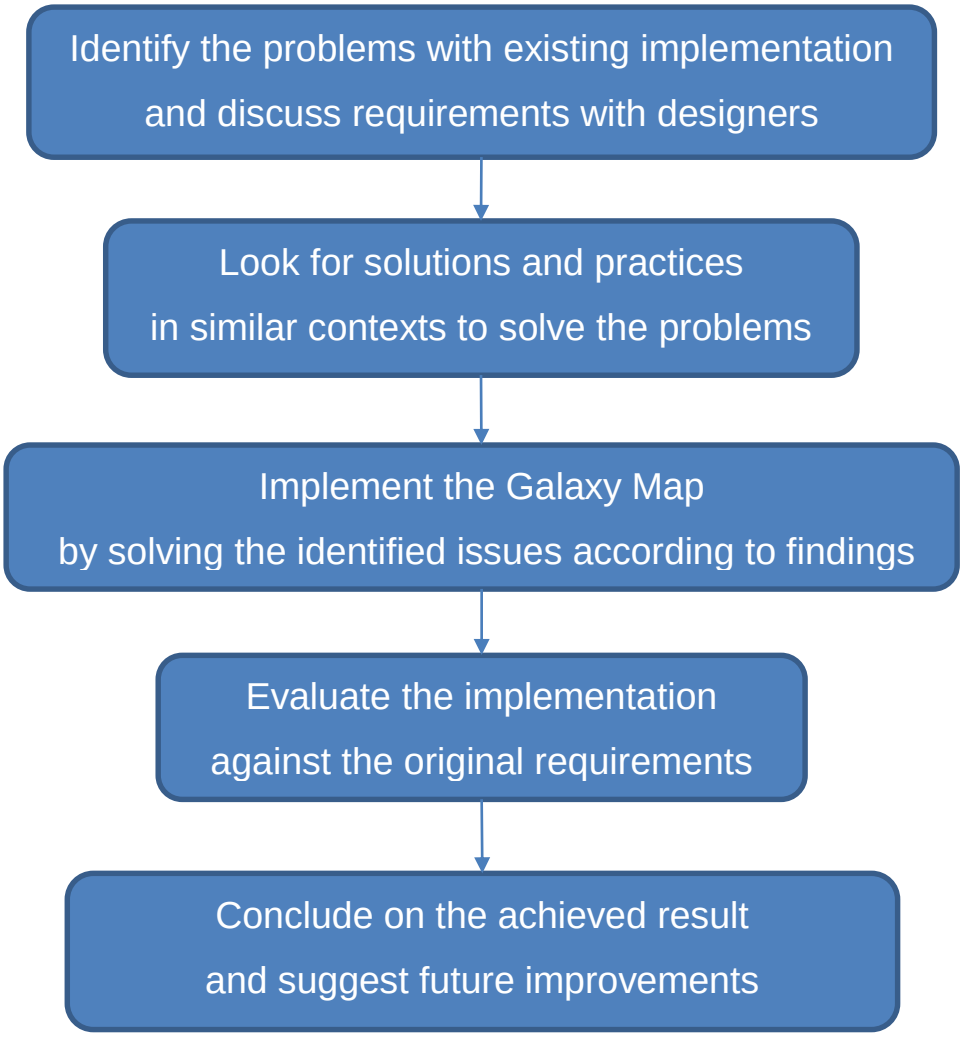


Figure 2. Research design flowchart

To clarify the details behind the steps in Figure 2, the study will specifically start by investigating on how the existing implementation of virtual world generation works. Discussion with Vulpine Games designers will be carried out at the same time. Brain-storming sessions will be arranged to make sure the gameplay is interesting with certain requirements while still technically possible to implement and finalize into the final release. The process will also cover open world design principles and how they affect the quality of the game from the player's viewpoint, which will later lead to the stage where major problems of the current existing implementation are recognized.

The next important step before the problems can be solved is the literature review to discover the solutions. This involves searching for similar issues in similar contexts where the solutions or suggestions exist. It will mainly be done in a fashion that several good alternatives are presented and weighted against each other for an optimum choice given the current situational problems.

Transitioning to the implementation phase is where continuous efforts of solving various problems to implement the map are detailed. It is also where communication with designers and the backend programmer become crucial to fine-tune various parameters that affect how the map looks and how the network traffic is handled.

At the end of the implementation, the Galaxy Map system is then evaluated against the original requirements on both functionality and performance grounds. For this, feedback from designers will be carried out and measurements will be done to ensure the backend system can cope with a large user base where there are thousands of concurrent players. The map at this phase can be fine-tuned according to the evaluation using certain parameters to better prepare for the launch.

The system will be ultimately concluded with a review of what was done and what was not done, unveiling the obstacles that prevented the design from being perfectly implemented and what effects they might have on the game. Certain potential improvements will be suggested as the result of discussion with designers to improve the future of the game.

3 Current State Analysis

This chapter will give an in-depth analysis of the current existing Galaxy Map implementation in Last Planets by first briefly introducing the background technology and architecture. The implementation is then explained by going through what has been done, what is still missing, followed by a glance on how virtual world generation is generally done in the video game industry before pointing out the major challenges that must be overcome to attain the design requirements. The chapter is concluded with the key findings from the analysis which define what needs to be done in the next stage.

3.1 Background Architecture and Technology

As pointed out in Section 1.1, Last Planets is a massively online multiplayer game for mobiles, where thousands of players can connect at the same time and interact with each other through a virtual world platform called the Galaxy Map. For this to work, the consumer device is installed with a client application or game, built using the Unity3D engine which supports all popular mobile target platforms including iOS and Android. Every time player starts the client, it will connect to the backend server which serves all the necessary data for the client to function and to display. This means that the client will send a TCP (Transmission Control Protocol) request towards the server to ask for the needed data at certain points during the game session. The connection and all requests to the server are handled by Photon Engine, a networking plugin for Unity3D.

As shown in Figure 3, the server side is comprised of several different servers, each having its own purpose. The server system utilizes Azure Cloud infrastructure to run virtual machines where Photon Server can be run, in correspondence to Photon plugin on the client side. The system basically stores all player data in a storage which can be retrieved through a request by the client.

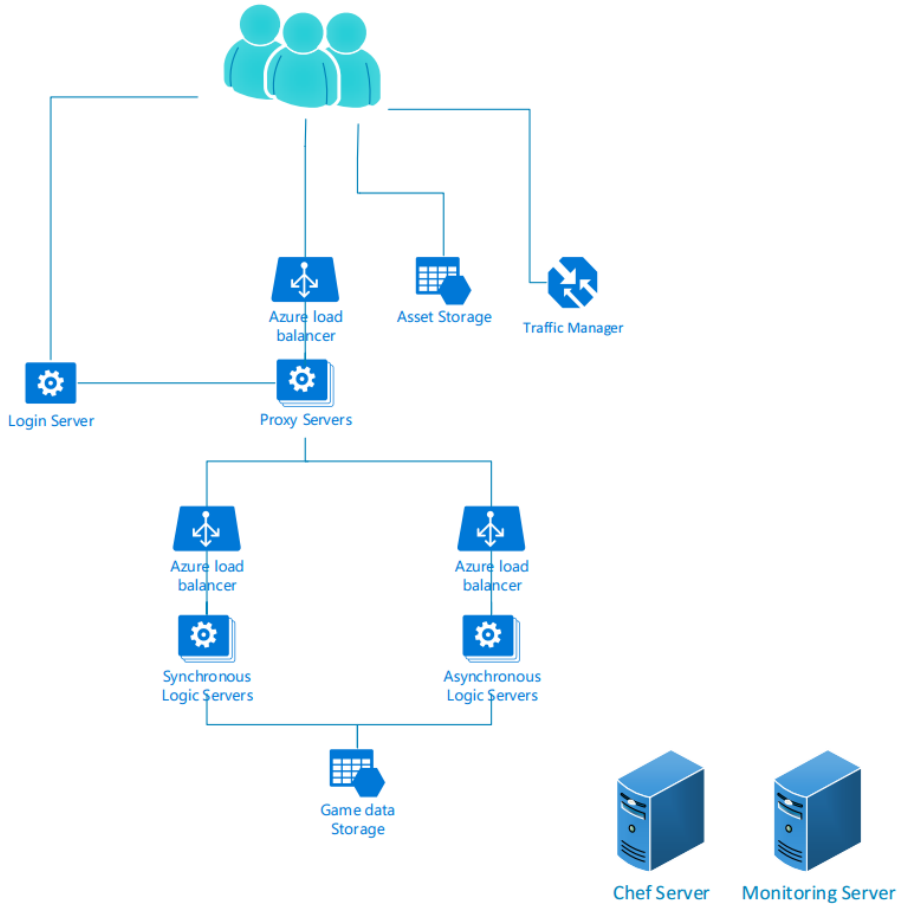


Figure 3. Last Planets server architecture

Other features and functionalities provided by the server, as depicted in Figure 3, include a login system, load balancing mechanism, logic checking system, monitoring system, logging system, etc., most of which are needed for the game to function. However, they will not be discussed in this study.

3.2 Existing Galaxy Map Implementation

The game is structured into four different scenes: build scene, map scene, view scene and attack scene. As the names suggest, the build scene is where the player starts. It is the home planet of the player, where building and management can be done as part of a typical base defense game. By going away from the home planet, a player can visit the Galaxy Map, where he/she can browse the entire in-game universe. From the map, other planets, either computer AI (Artificial Intelligence) or a human player, can be discovered, scouted and attacked, based on the player's demand.

As described, the Galaxy Map plays an important role in the game, where it acts as a bridge to connect players to the rest of the virtual universe. Other parts of the game cannot be developed and reliably tested as the same time by other developers without the existence of the Galaxy Map. At the same time, the map system has many requirements that need to be fulfilled for production, which cannot be done within any short and reasonable timeframe. For this reason, a temporary solution was used to accommodate the need.

The placeholder implementation of Galaxy Map was made and has been serving the purpose of interconnecting different parts of the game for testing. However, the state of the map is far from what needs to be achieved. The system simply divides the universe into a rectangular grid consisting of equally sized squares. Each planet in this system takes up one square and has its exact position being randomized within that square. Each alliance occupies a bigger area, which is designed to be a composition of 16 smaller squares. An alliance is a mimicked model of a solar system where players can gather together as an ally to overcome challenges in the game.

The design is rather simple, with no reliable or exact coordination system to identify any targets in this universe. It also only supports the planet and alliance, disregarding any other types of object the game may have. In addition to those problems, the biggest issue is the fixed, rigid size of the universe, which is 50 squares in height and 50 squares in width. The issue stems from the fact that the whole universe is always created with that size and only supports that size. Every time the client is started and the map is accessed, the game will create over 2000 objects (planets and alliances), causing a big stress on the memory usage, with the biggest concern being low-end devices.

In an ideal scenario for the backend server, the universe is empty and there are almost no players, the universe will be filled with mostly AI players. Those AI opponents are generated on the client side, thus requiring no data from the server. On the other hand, if the map is filled with players, there would be more than 2000 player objects to retrieve from the backend. From the bandwidth utilization viewpoint, it is a bad practice. From the player's viewpoint, it can be a long waiting process for the client to fetch the data. The cost of the operation will also prevent the game from being able to refresh the state of the map dynamically to reflect the most up-to-date data.

In either cases, be it server-side or client-side, the waste in processing power, memory usage and bandwidth usage, while the game only allows the player to view a small portion of the map at any given points, is an unacceptable weakness. This reason, when coupled with the fact that there are no possibilities to expand the user base beyond 2000 players, makes the game a failure as a massively online multiplayer game.

3.3 History of Procedural Generation in Video Game

Procedural generation is not new in the video game industry. As pointed out in the introduction, it is a method of generating data using a computational algorithm, instead of manually creating them by hand. By using the method, any items, objects, terrain, etc. can be created by the computer. This does not only save time for the content creation process, which is vital for a small developer team, but also allows a greater degree of randomness or variety while retaining a small storage usage.

There have been many games employing the technique to create in-game contents. The practice can be traced back to 1978, with Beneath Apple Manor as an example. The game is a RPG (Role-playing Game) video game, which generates its dungeon level randomly every time a new game is started. (My Abandonware 2016.)

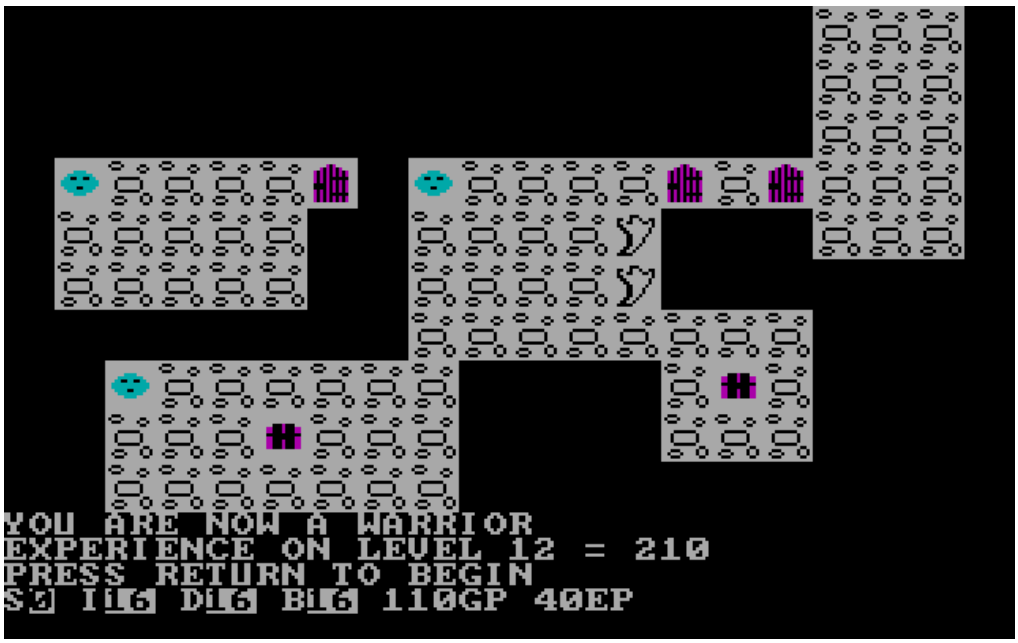


Figure 4. A randomly generated level in Beneath Apple Manor
(Copied from My Abandonware 2016)

As seen in Figure 4, Beneath Apple Manor is mostly a text-based, ASCII (American Standard Code for Information Interchange) video game, running on a system with limited memory constraints. This is understandable as the practice not only saves the time it takes to create those levels, but is also used to coping with the limited system resource at the time. The created level does not always have to be completely random. A good example in the early days would be Akalabeth, which can use a seed for the random-number generator, chosen by the player to create the entire level (Maher 2011).

Moving on to the modern time, Minecraft is amongst the most popular video games based on procedural generation to create its own virtual world. Given the same seed, the algorithm will always result in the same world. However, the world in Minecraft is much more advanced, as well as being unlimitedly rich. It is divided into chunks which are generated and stored as the edge of the world is reached.

The most recent attempt to utilize procedural generation that attracts interest around the world is No Man's Sky. The game uses a similar technique as Minecraft, but more complex through the extensive usage of mathematical equations which result in an enormous and diverse galaxy containing 18 quintillion planets, each having its own biome. (Khatchadourian 2015.)

Figure 5. The various generated world in No Man's Sky
(Copied from Khatchadourian 2015)

Figure 5 shows the potential of what procedural generation algorithm can do, to the extent of creating an entire unique universe that makes sense to human eyes. This is done purely by code but yet retaining the feeling of an artistic touch.

3.4 Requirements for Last Planets' Galaxy Map

In the case of Last Planets' Galaxy Map, the scale will not be as large and as ambitious as either No Man's Sky or Minecraft. The focus will be on the capability to allow thousands of players to appear on the map, where new players are constantly joining the game and placed at a specific location, determined by an algorithm in the backend. At the same time, the performance on the client need to be steady and easy to navigate to allow a pleasant exploration experience, while keeping network traffic to a minimum without compromising the visual data on the map, thus allowing the player to freely interact with different types of objects on the map.

The Galaxy Map should be divided into clusters, each cluster being divided into areas, and each area is again cut into small slots where galaxy objects can be placed. Each cluster will carry a unique identifier which is served as a seed to create the content inside the cluster, which includes all the planets, alliances, events, background nebulas, etc., to bring the universe to life. Any object that occupies a whole area on the map needs to maintain a minimum distance to other similar objects, so that they are not too close, while at the same time maintaining a non-uniform and interesting pattern, without giving a hint that they have been generated computationally.

The data on the map should be pulled from the server progressively as the player goes. The same applies to creation of objects on the map, in which only objects within the player's view are generated. This practice is required to make sure the game is run on lower-end hardware, saving server bandwidth and resources to accommodate more concurrent online players. The data fetching mechanism should be dynamic and controllable using certain parameters to allow data requests only when a specific zoom level is met or only when data is determined as outdated, thus utilizing caching for the purpose of data refresh without compromising bandwidth usage.

The universe needs to be prepared for expansion, through the process of adding more clusters to the existing world, as well as cluster reordering features in near future. This guarantees that inactive clusters will be moved to the outer edge, keeping the active players at the center along with new players to boost the social gameplay experience.

3.5 Key Findings from the Current State Analysis

While the Galaxy Map's requirements do not indicate the need to generate a universe that matches the level of complexity of other reviewed video games in Section 3.3, it is clear the current temporary solution has not managed to implement the required features. Before the discussed requirements can be researched and implemented, a list of conclusions on the major challenges based on the analysis, is made:

- Coordinate system and procedural generation
- Unique random pattern
- Dynamic object generation
- Effective network resources usage
- Support for various object types.

Each topic in the list will be discussed in the next chapter, with extensive study on how they can be solved based on contemporary literature. Some topics can be discussed where multiple approaches can be weighed against each other to determine the best solution in the context of Last Planets. The knowledge will then be used to finalize the technical design for the final implementation.

4 Theoretical background

This chapter provides the details of all major topics that were considered in Chapter 3.

4.1 Coordinate System and Procedural Generation

In any video game, the usage of a coordinate system is vital to the capability of rendering, placing and animating objects. The Cartesian coordinate system is a typical choice amongst video game programmers. This involves the representation of the virtual world using a coordinate tuple of either x, y in 2-dimensional world or x, y, z in 3-dimensional world. Any values that follow the coordinate system can be used to define the position of an object within the world space or the local space, which is relative to another object's coordinate system. (Wolf et al. 2008.)

For any procedural generation algorithm, a well-functional coordinate system is needed for the algorithm to work. The process will utilize the system to generate geometries, objects, terrain, etc., in places where they make most sense in relation to its surroundings according to the specified parameters. One simple example of how procedural generation works is Dwarf Fortress. The game is basically an ASCII or text-based game and is originally coordinated using a simple single 2D (two-dimensional) plane.

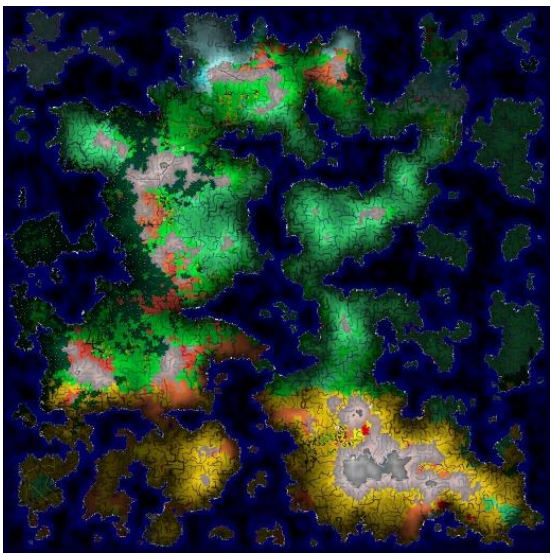


Figure 6. A world generated procedurally in Dwarf Fortress
(Copied from Champandard 2012)

Figure 6 visualizes the generated world map in Dwarf Fortress, where each point in the coordinated world carries terrain information. The information is not generated randomly for the point alone, but rather as the result of multiple simulation layers being applied to the entire world map. The simulation layers include vegetation, weather, temperature, etc. The realism is added to the world by using the element of history, which is simulated during the generation process, resulting in virtual historical events, creating villages, roads, cities, as signs of civilizations. (Champandard 2012.)

A more advanced example is Minecraft, as the world is established in 3D (three-dimensional) coordination system with the x-axis denoting the longitude, z-axis representing latitude, while the z-axis depicting the elevation. Each block in the world equals one coordinate number while each chunk consists of 16 blocks wide and 16 blocks long, covering the entire spectrum of height. (Minecraft Wiki 2016.) The terrain is randomized using the Perlin noise algorithm, which basically functions similarly to the simulation layers from Dwarf Fortress as discussed above. The process is refined over many noise layers to enrich the environment. (Fingas 2015.)

According to the requirements, the Galaxy Map is a simpler world compared to both Dwarf Fortress and Minecraft, where no terrain information exists. Instead, the map is decorated with various object types that do not overlap each other, except for the background nebulas. The map can, therefore, use a divided coordination system similar to that of Minecraft, but in 2D format, to contain each object on the map within its own domain.

In correspondence to the chunk in Minecraft, the biggest division unit in the Galaxy Map will be a cluster, with each cluster housing 256 areas, sized 16 areas wide and 16 areas long. The reason behind 256 areas choice is that only 1 byte will be needed to number all the areas, thus saving the network traffic but still having a reasonably large enough division. To allow a cluster to contain more objects and to better organize the data, another layer of division is added underneath each area. The area is then divided in 16 slots, sized 4 slots wide and 4 slots long, with the reason of being a rational choice for the scale of the universe. However, not all areas and slots will be occupied as discussed in Section 4.2 below.

4.2 Unique Random Pattern

Humans and animals in general are known for the ability to use spatial memory to navigate through the environment. But how does this behavior function in a virtual world? A study was conducted by Newman et al. to examine the human's capability to navigate in a virtual world. The research used several towns designed specifically with roads and stores in a virtual reality, where participants had to take part in certain trials. The results indicated that subjects were able to navigate within the created world just as in real life, with preference leaning towards the landmarks instead of the layout. (Newman et al. 2007.)

The study above clearly points this section towards an interesting direction, which involves solving how a virtual world should be designed. There are many algorithmic ways to populate a virtual world. In a simple 2D world like the Galaxy Map, the simplest methods that can be used are either to randomly place objects within the boundary of the world or to place them on a grid with a random offset. They are both good solutions since it is relatively easy to put them into practice as pointed out by Herman Tulleken. (Tulleken 2009.)

However, the result may not be desirable, particularly when we want players to be able to navigate the map. To put this into perspective, we want players to recognize the pattern of the stars (planets) as a layout and the uniqueness of the bigger entities (alliances, events) as landmarks.

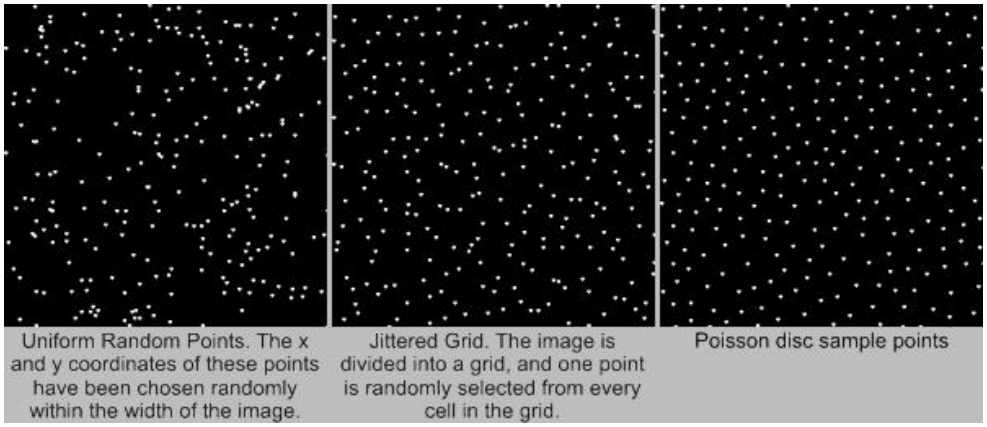


Figure 7. Different types of placement algorithms (Copied from Tulleken 2009)

As shown in Figure 7, several placements strategies are depicted side-by-side for comparison. The first technique on the left distribute random points within the image, disregarding the possibilities of overlapping, in return, the outcome looks unordered and chaotic, which is relatively natural. The image at the center shows how positioning using a grid can be done. The world space is divided into a grid of equally sized cells, where each cell can contain a point. The point is then placed randomly within the cell, effectively create a sense of natural randomness, while preventing points from overlapping each other. Finally, the Poisson disc sampling method yields a result in which all objects are randomly placed, but each object still remains at a minimum distance to the surrounding objects. This method solves both the overlapping and the minimum distance problems, which ideally matches with the requirement for big objects within the Galaxy Map.

There are many existing algorithms that can be used to produce a Poisson disc sampling distribution of points. One of the reasonably fast algorithms that can be run with O(N) complexity, with N being the number of disc samples, is proposed by Robert Bridson. The algorithm works by procedurally adding new points surrounding the existing points, as long as they satisfy the check of the minimum distance requirement. Even though we are just looking for a solution in a 2D space, the method can be extended for use in arbitrary dimensions, thus making it worth mentioning as a reference for other projects. (Bridson 2007.)

However, from a design standpoint, it is easily noticeable that the Poisson disc sampling method results in a uniform and artificial look. This potentially leads to a dull experience for players who are keen on exploration, as there are no clear patterns emerging from their observation after they spend time with the map. This takes us back to the previous discovery of how the human spatial memory is used to navigate. The need to create patterns like roads and landmarks suggests that we need to emerge those patterns from the generated space to assist players with navigation and memory attachment.

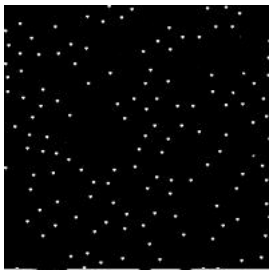


Figure 8. Custom sampling technique (Modified from Tulleken 2009)

Without modifying the Poisson algorithm, a simple solution can be devised to solve the issue. After all points are processed and placed by the Poisson algorithm, a post-processing algorithm can be used to run through all the placed points and remove them randomly across the domain. Figure 8 illustrates a hand-made result of the technique, which looks a lot more organic when compared to the original solution. This can be related to the previously explored formulas used by Minecraft and Dwarf Fortress, where multiple layers of refinement are applied to get the final effect.

For smaller objects in the Galaxy Map, which are planets, there is no strict requirement on the minimum distance between themselves as a planet alone does not inhabit an entire area. In addition to that particular reason, there are more planets than big objects (alliance, event, etc.). Thus, executing a simpler algorithm on planets is more ideal to solve the distribution problem. With that said, the mentioned jittered grid placement method suits the situation quite well. However, with a large number of planets being spread within the map, the overall universe would be exceptionally dense. An overly crowded world can introduce a navigation issue to the player, as it makes the pattern and the neighborhood less memorable. Nevertheless, the problem can be easily addressed by employing the same post-processing technique mentioned above to remove the excess of planets.

4.3 Dynamic Object Generation

As concluded in Section 4.2, the map must not produce all objects in the world when the map is accessed. It is an example of bad resource utilization as only a portion of the map can be viewed at any instance of time, not to mention the infinitely expandable nature of the map. The solution to this is to generate the object dynamically as the player explores the map. When the player first enters the map, only his/her home area and a limited number of surrounding areas will be generated on the client device. As the player pans towards a certain direction, the created objects that are situating in the opposite direction will be destroyed to make room in the client's RAM (Random Access Memory) for newly generated ones.

Most games have a spawning system of their own, which governs how objects are created and destroyed, depending on when they are needed. The main concern regarding this system is the dynamic nature of memory allocation and deallocation, which can be relatively slow. The problem is aggravated due to the difference in size between object types, which can potentially cause memory fragmentation, making allocation slower and harder. (Gregory 2014, 904.)

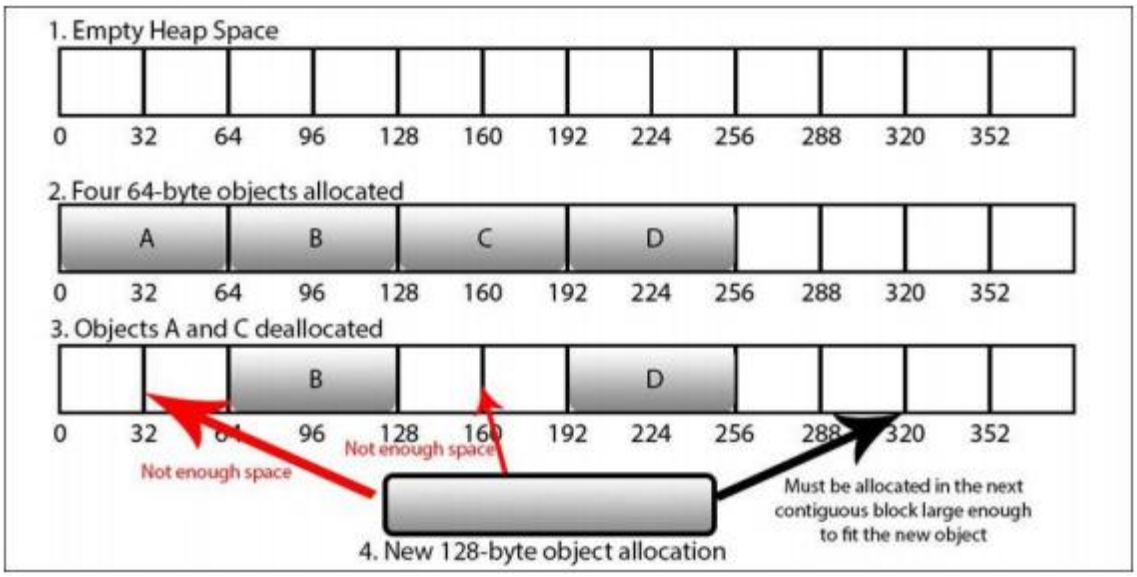


Figure 9. A simple memory fragmentation scenario (Copied from Dickinson 2015, 199)

Figure 9 demonstrates how memory fragmentation can affect future memory usage. Consider an empty space in the memory heap. In this case, a heap is a dynamic memory space, as oppose to a static, limited, pre-allocated space in the memory called stack. After four 64-byte blocks of memory have been used, freeing two of them (block A and block C) leave the memory with two separate free 64-byte memory blocks. Since they are not contiguous, allocating another bigger block (128-byte for example) later on would not be possible at either location where block A and block C were. (Dickinson 2015, 199.)

The heap memory will naturally be filled with more of those small, unusable blocks over the time if there is no management technique employed. Moreover, causing a premature out-of-memory situation where there is no contiguous memory block for new allocation is just one issue. A new contiguous block for allocation will take a longer amount of time to be found as well, as it result in a more demanding look-up work for the system. (Dickinson 2015, 199.)

There is another example of how the allocation process can take a severe performance hit. Every time the game makes a request for a free memory block, the CPU (Central Processing Unit) will have to search through the application memory space for a free and contiguous space. If not, the garbage collector (in a managed language, for example C#) will have to run through every single object reference in the memory to mark which objects are no longer used. Unused or no longer referenced objects will be deallocated to make room for the requested free block. If there is still no available space for the request, a new heap memory block will be asked from the OS (Operating System) to accommodate the demand. (Dickinson 2015, 200.)

The depicted scenario above shows how much work the CPU has to handle to satisfy the request. This is likely to create hiccups or freezing time while the game is running, causing frustration to the player. The garbage collecting process is even more demanding as the size of the heap grows, potentially leading to a much worse performance. For this reason, smart usage of memory allocation and deallocation is needed. Also a simple method to reduce GC (Garbage Collector) frequency is to call it manually when user experience matters least, for example, during level loading time. (Dickinson 2015, 200.)

It has been shown until this point that the allocation and deallocation frequency should be avoided as much as possible. However, the Galaxy Map still needs a dynamic spawning system which spawns and removes objects on demand. One good technique that is commonly employed is using a pool for objects. The method pre-allocates a pool in the memory that can contain as many objects as needed. However, all objects that exist in this pool are hidden from the player. Every time an object is required, it will be taken away from the pool and become visible. If the object is no longer in use, it will then be put back into the pool and is hidden away, in other words, it will be recycled. (Dickinson 2015, 220.)

The practice can be perfect if the designer knows exactly how many objects are needed for the pool. In that case, after the pool is initialized, there will be absolutely no extra allocation or deallocation happening while objects are dynamically spawn. Memory usage will be highly controllable and expected, guaranteeing no incidences where the game may freeze and crash due to a memory issue. (Gregory 2014, 904.) However, with the Galaxy Map supporting different levels of zoom based on the player's choice, the number of required objects in the pool may vary. A closer zoom level requires fewer objects and vice versa. Further zoom will need more, thus raising the demand for having

a more sophisticated pooling system in a strict memory constraint scenario. Another notable issue is the variety of object types, which may require different object pools, discussed in Section 4.5.

4.4 Effective Network Resources Usage

Before strategies for network resources usage can be discussed, a comparison between TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), as well as the reason why TCP was chosen needs to be reviewed. For any networking game, the developers need to choose for themselves an appropriate networking protocol. The networking protocol is used to transfer packets over the network and it is relatively important that the correct method is used. As pointed out by Christoffer Lernö, the general opinion is that action games typically use UDP as their transfer protocol, but then for action MMO (Massively Multiplayer Online) games like World of Warcraft, TCP is used. (Lernö 2014.) This denotes a hint that different kinds of games may require a different protocol. Glenn Fiedler confirmed this by mentioning that the choice is based completely on the game being made. (Fiedler 2008.)

The TCP protocol governs mostly everything happening on the internet, as it is a simple and reliable protocol. The simplicity stems from the TCP way of functioning. It basically forms a connection between two receiving ends, then transmits the data similarly to writing or reading a file. The transmitted data is assured to reach its destination since the protocol is reliable and ordered. Another positive point about TCP is the support for arbitrary packet size and the ability to split a packet into smaller packets to allow data streaming, making it an out-of-the-box solution for basic needs. (Fiedler 2008.)

Nevertheless, a good implementation on top of TCP is not feasible to be made, as there are various issues that need be addressed, such as packet congestion handling, disconnection handling, DoS (denial-of-service) attack, etc. The most problematic property of the TCP protocol is, however, the congestion control mechanism. Every time a packet is loss during a TCP transmission, it is interpreted as bandwidth limitation and therefore throttling the number of packets to be sent. This is especially damaging to a wireless network connection, as any dropped packet will signal the congestion control to throttle the transmission, causing a huge delay to the process. (Lernö 2014.)

The UDP protocol is, as opposed to TCP, a rather difficult protocol to use. The protocol does not offer ready-made experience that TCP offers. There is no connection, but a packet is just sent directly to the receiver. When accompanied with the unreliable nature, where receiving packets are sometimes lost and unordered, UDP protocol appears to be the least favorite choice. Despite the negative points, UDP is actually a very simple protocol underneath, with nothing being fixed. This allows a huge amount of freedom when it comes to implementation. The developer can choose to manually implement and customize most aspects of the protocol to fit the requirements, thus enabling an entirely different possibility of how a game can be done. (Fiedler 2008.)

Given such customizability power, however, does not mean every game has to use UDP over TCP for an optimum networking performance. Action games mostly have a very high demand for real-time response, thus making it reasonable to use UDP to satisfy the experience. World of Warcraft is one example of an action game where TCP works without having to resort to UDP. The reason for this is simply due to the lack of need for a strict in-game coordinate targeting and processing. Most operations can be done using the target ID instead of using a precise coordinate, which fluctuates constantly as the player moves. Another thing that can be noted is the use of animation in this game, meaning that any delay in network operation can be easily masked and player would still not realize the latency. (Lernö 2014.)

In the case of Last Planets, there are not many animations to mask the lag during network transmission, but there is also no harsh demand on real-time experience either. Most operations on the client can be done while the network requests are sent in the background, similarly to World of Warcraft. Meanwhile, the client continues to display data calculated locally, with almost zero or very little interruption to the experience. For the provided reason, coupled with the ease of use TCP provides, TCP was chosen over UDP for the game.

Unfortunately, while most of the requests over TCP for the game can be done without much consideration, there is an exception. The most demanding case for data availability is the Galaxy Map, where all players in the game need to be laid out in the universe in the most real-time or most up-to-date manner as possible. This could mean that the world data have to be constantly fetched from the server to keep the state of the map fresh. In the real world, that is not possible, due to the nature of TCP throttling packets sending as soon as some packets are lost or dropped. To make it clear, the more requests being

made by the client to the server, the higher the chance to have a least one packet dropped. It is also worth mentioning the natural cause of packet lost due to real bandwidth limitation, as thousands of players can request map data at the same time.

To combat the issue, several strategies have to be used in this particular scenario. For the client side where requests are made, the main method would be to reduce the number of requests. In order to achieve this, one simple technique that can be used within the map is to group as many requests as possible into one. Each time the player pans around, there will be some areas where data is needed so they can be displayed. Instead of sending a request for only one area or a few areas as soon as they are found, the game can accumulate the number of areas to a large enough number and send the request at once. The accumulation can be fine-tuned with a parameter so it does not interfere with the player's experience. This should be effective enough to drastically reduce the number of redundant small requests. However, we also need a method to cope with the requirement of up-to-date data reflection, without sending a constant stream of requests for the areas that are just requests.

Caching is a very good solution to introduce delay interval between requests of the same area. According to Venkatesh, caching is a way to prevent the repetition of an operation by storing the data so it can be used at a later point. Before any request to retrieve the data is made, the local data storage will be checked to verify whether or not the data exists in the cache. If it does, it will be considered a cache hit, otherwise a cache miss. Each time a cache miss occurs, a request will be forced to be made to retrieve the data, after which it will be stored in the cache for future reference. (Venkatesh 2014.)

Data in the cache can, however, be outdated. To meet the requirement in the Galaxy Map, the cache for outdated areas is then designed so that every reference to an outdated area will be considered a cache miss. This effectively forces the client to fetch the latest data for the referenced areas, while still ensuring there is an interval between the current request and the last time the request is made.

To determine the optimal interval between requests and still guaranteeing the status of the data, a discussion with Vulpine Games designers was held to figure out the most vital area types that need the most updates. The outcome suggests a highest priority on event areas, with alliance areas coming in second, whereas areas filled with only planets receive least attention. The parameters to adjust those intervals are then required to present in the configuration file, which can be supplied on the fly every time the client connects to the server.

4.5 Support for Various Object Types

The Galaxy Map is designed to support various object types that can be spawn dynamically. The most simple and traditional approach is to create a new generic object at the specific position for that object, generated by the procedural generator. After the data for the object is retrieved, either from the server or from the cache, another object will be created depending on the object type through a switch. The placeholder object will then be destroyed and this new object will replace it. The first problem with this simple solution is that every time a new object type is introduced or changed, the switch structure will have to be changed. The structure itself, as a switch containing object creation instructions, is cumbersome as well as being difficult to maintain. This is not to mention the lack of versatility of reusable code.

The factory method pattern can be used in this case to abstract the reusable code elements that are shared between different object type creation instructions. The result is a more compact and maintainable code base. (Freeman 2004, 134.)

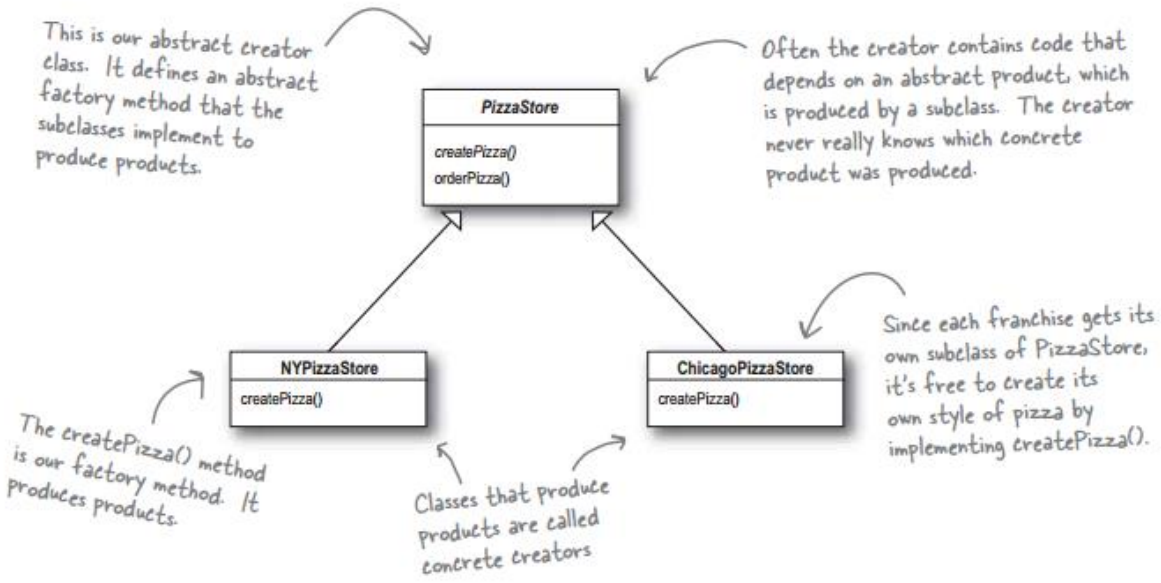


Figure 10. An example of factory pattern method usage (Freeman 2004, 131)

A demonstration of factory pattern usage is shown in Figure 10. PizzaStore is an abstract class which different PizzaStore classes can inherit from. Contained within PizzaStore is the createPizza abstract method. This method is overridden by the derived class to customize how pizzas are made from the store. (Freeman 2004, 131.)

When applied to the Galaxy Map, the PizzaStore can be called GalaxyObjectShape and correspondingly, classes deriving from it can be AllianceShape, EventShape, etc. The derived classes can hold instructions to create any object of the corresponding type. As discussed in Section 4.3, we will use the object pooling method to reduce the frequency for allocation and deallocation of objects. This means that the mentioned classes can act as object pools at the same time. However, another important matter that we also discussed in Section 4.3 is the memory fragmentation issue, caused by the allocation of different sized memory blocks. Having different pools which instantiate different sized objects may lead to fragmented memory later on if some pools adjust their size dynamically to cope with the hardware constraint.

To cope better against the issue, the need for a single generic object type arises. The generic object can have a structure and all the necessary parameters needed so it can be shaped or modified into any desired object type in the game. This solves the potential memory fragmentation issue by using a single shared pool between all objects of the same size, as well as eliminating the need to create a placeholder object before the data arrives. (Gregory 2014, 906.) By using this improved approach, classes derived from GalaxyObjectShape will no longer create objects on their own. Instead, when an object is created, it will be taken out from the shared pool as a visible placeholder object on the map. Once the required object data is retrieved, the class corresponding to the object type will take the object and its data as parameters to shape it according the internal instructions within the class.

5 Implementation

In this chapter, the implementation of the Galaxy Map regarding the most challenging aspects are detailed. This includes the specific design and technical workflow to achieve the design based on the literature review and conclusions in Chapter 4.

5.1 Procedural Generation of the Unique Pattern

For the purpose of procedural generation and dynamic object spawning, the precise coordinate system used in Last Planets is as discussed in Section 4.1. The map will be divided into an unlimited number of clusters. Clusters can be patched together seamlessly to form the virtual world in the game.

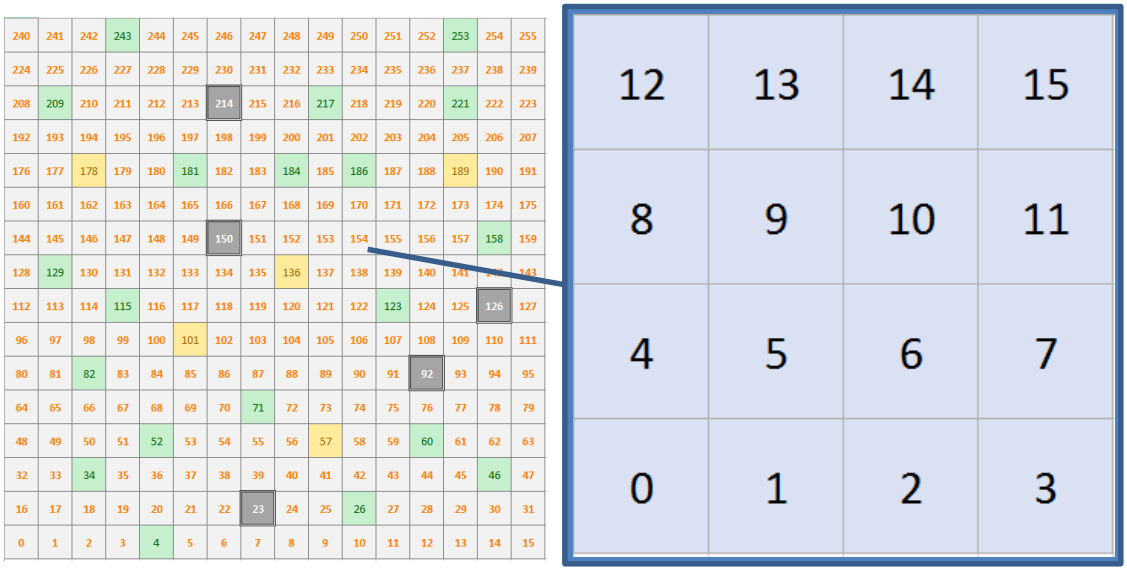


Figure 11. Cluster, different area types within a cluster and planet slots within a normal area

On the left side of Figure 11 is a cluster which is divided into 256 numbered areas of different types. The green squares represent alliance areas, whereas the yellow ones are used as event areas, the grey ones are empty while the rest are normal areas. Each normal area is then divided into 16 smaller slots or cells to accommodate planets, either for player or AI, as illustrated on the right side of Figure 11. Areas within a cluster are numbered from 0 to 255, while slots in an area are indexed using numbers from 0 to 15, starting from the bottom left. Both numbering systems exist to aid the object management along with the conversion to and from the universal Cartesian coordinate which is used by the dynamic spawning system.

The coordinate system parameters have been designed to be changeable through the configuration file. Various tests and iterations have been gone through with designers to determine the best option. The mentioned values are final and are unlikely to be changed in the released version of the game.

To guarantee the uniqueness of any cluster, a unique identifier GUID (Globally Unique Identifier) is used to label each cluster. GUID is a 16-byte data structure that holds a unique sequence of numbers that have a very low chance of collision or duplication. (MSDN 2016.)

```
"936DA01F-9ABD-4d9d-80C7-02AF85C822A8"
```

Listing 1. An example of GUID format

As shown in Listing 1, the GUID, while guaranteed to have a very low repetition rate, is not usable for the random number generator in Mono C#. The GUID is thus used to produce a 4-byte integer hash, which is then fed into the random number generator to produce the pattern within the cluster. To create a pattern, areas in the cluster will be first taken into consideration.

```
int allianceCount = Conf.Instance.GalaxyMapConstant.AllianceCount;
int eventCount = Conf.Instance.GalaxyMapConstant.EventCount;
int emptyCount = Conf.Instance.GalaxyMapConstant.EmptyCount;
int relayCount = Conf.Instance.GalaxyMapConstant.RelayCount;

int totalBigObjects = allianceCount + eventCount + emptyCount + relayCount;

float initialMinDistance = 3;
List<CommonVector2> bigObjectPositions;

while (true) {
    UniformPoissonDiskSamplerInt.SetSeed (seed);
    bigObjectPositions = UniformPoissonDiskSamplerInt.SampleRectangle (
            CommonVector2.zero,
            new CommonVector2 (
                GalaxyMapConfig.Instance.ClusterSize_AreaUnit,
                GalaxyMapConfig.Instance.ClusterSize_AreaUnit),
            initialMinDistance);

    if (bigObjectPositions.Count < totalBigObjects) {
        initialMinDistance *= 0.95f;
        if (initialMinDistance < 2) {
            break;
        }
    }
    else break;
}

// Trim to match totalBigObjects count
while (bigObjectPositions.Count > totalBigObjects) {
    bigObjectPositions.RemoveAt (rand.Next (0, bigObjectPositions.Count - 1));
}

return bigObjectPositions;
```

Listing 2. Generation of areas for big entities based on a seed

The generation process to create the pattern is first started by generating areas for big entities on the map, namely alliance area and event area. As detailed by Listing 2, the total number of those areas and their position will be generated through the usage of a Poisson disk sampling algorithm, implemented after Robert Bridson's method. Starting with a minimum distance of 3 areas between big entity areas, the solution is then iterated over and over again, each time with reduced minimum distance limitation until the number of needed areas match the requirement. The result might yield a little bigger number of areas than needed, thus the need to trim the excess at the end to produce the final result. However, the generation method only returns the coordinate of the reserved areas, due to the code reusability reason. To obtain the specific areas reserved for a specific area type, another method is used to filter out those areas accordingly.

```
int toBeRemoved;
var reservedAreaPositions = GetReservedAreaPositions(clusterId);

// Trim out and return Event Areas if neeeded
var eventCount = Conf.Instance.GalaxyMapConstant.EventCount;
var eventObjects = new List<CommonVector2> ();
int targetNumberOfObjects = reservedAreaPositions.Count - eventCount;
while (reservedAreaPositions.Count > targetNumberOfObjects) {
    toBeRemoved = rand.Next (0, reservedAreaPositions.Count - 1);
    eventObjects.Add (reservedAreaPositions[toBeRemoved]);
    reservedAreaPositions.RemoveAt (toBeRemoved);
}
if (reservedAreaType == AreaType.Event) return eventObjects;

// Trim out and return Empty Areas if neeeded
var emptyCount = Conf.Instance.GalaxyMapConstant.EmptyCount;
var emptyObjects = new List<CommonVector2> ();
targetNumberOfObjects = reservedAreaPositions.Count - emptyCount;
while (reservedAreaPositions.Count > targetNumberOfObjects) {
    toBeRemoved = rand.Next (0, reservedAreaPositions.Count - 1);
    emptyObjects.Add (reservedAreaPositions[toBeRemoved]);
    reservedAreaPositions.RemoveAt (toBeRemoved);
}
if (reservedAreaType == AreaType.Empty) return emptyObjects;

// Trim out and return Relay Areas if neeeded
var relayCount = Conf.Instance.GalaxyMapConstant.RelayCount;
var relayObjects = new List<CommonVector2> ();
targetNumberOfObjects = reservedAreaPositions.Count - relayCount;
while (reservedAreaPositions.Count > targetNumberOfObjects) {
    toBeRemoved = rand.Next (0, reservedAreaPositions.Count - 1);
    relayObjects.Add (reservedAreaPositions[toBeRemoved]);
    reservedAreaPositions.RemoveAt (toBeRemoved);
}
if (reservedAreaType == AreaType.Relay) return relayObjects;

// The remaining areas are reserved for alliances
return reservedAreaPositions;
```

Listing 3. Filter method used to extract specific area type out of reserved areas

The method shown in Listing 3 uses the same seed that was used for the previous areas generation method. This persistent seed is then used to go through a process of deterministic trimming as depicted in Listing 3 to get the exact same result every time. The outcome received from this method is then used to build a cache on the client which holds the coordinate data for all the areas in the cluster as well as storing all the network data for all objects contained within.

By using this implementation, each cluster is now populated with five area types: normal area, alliance area, event area, relay area and empty area. The normal area will be used to house planets as designed, whereas the alliance area will be used to contain a single alliance object, with event area behaving similarly. The relay area is reserved for future usage as relay feature is not yet implemented, while the empty area is simply added to enhance the randomness and uniqueness look of the cluster. Since the trimming is random against the reserved areas, the pattern is then improved further as illustrated in Figure 11.

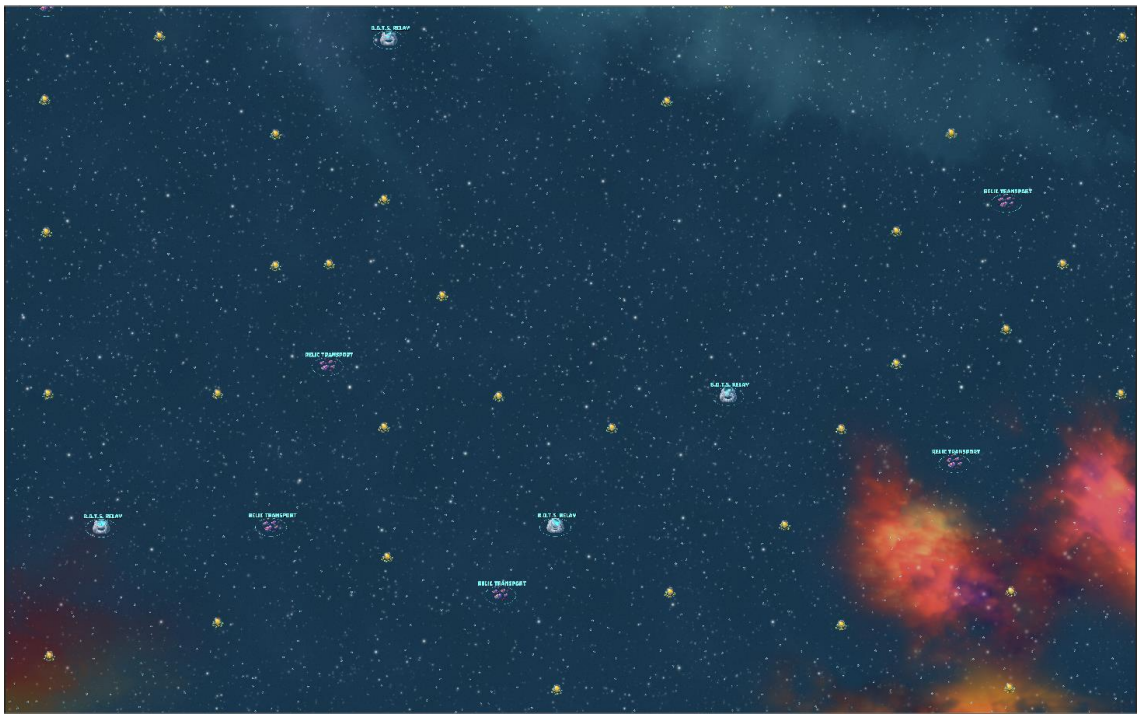


Figure 12. A cluster populated using the implemented procedural generation technique

Figure 12 introduces the first look at a fresh cluster of the implemented Galaxy Map, with the yellow dots being alliance seeds, which a player can create an alliance upon, along with different event types and empty areas randomly distributed in between. The combination of different areas, distributed with a minimum required distance clearly created a relatively natural and unique looking universe. This effect will be greatly signified when it is populated with alliances, established by players randomly across the universe. While the pattern problem is solved on the large scale, when zoomed in closely to examine the normal areas where planets reside, the close-up pattern issues emerge evidently in Figure 13.

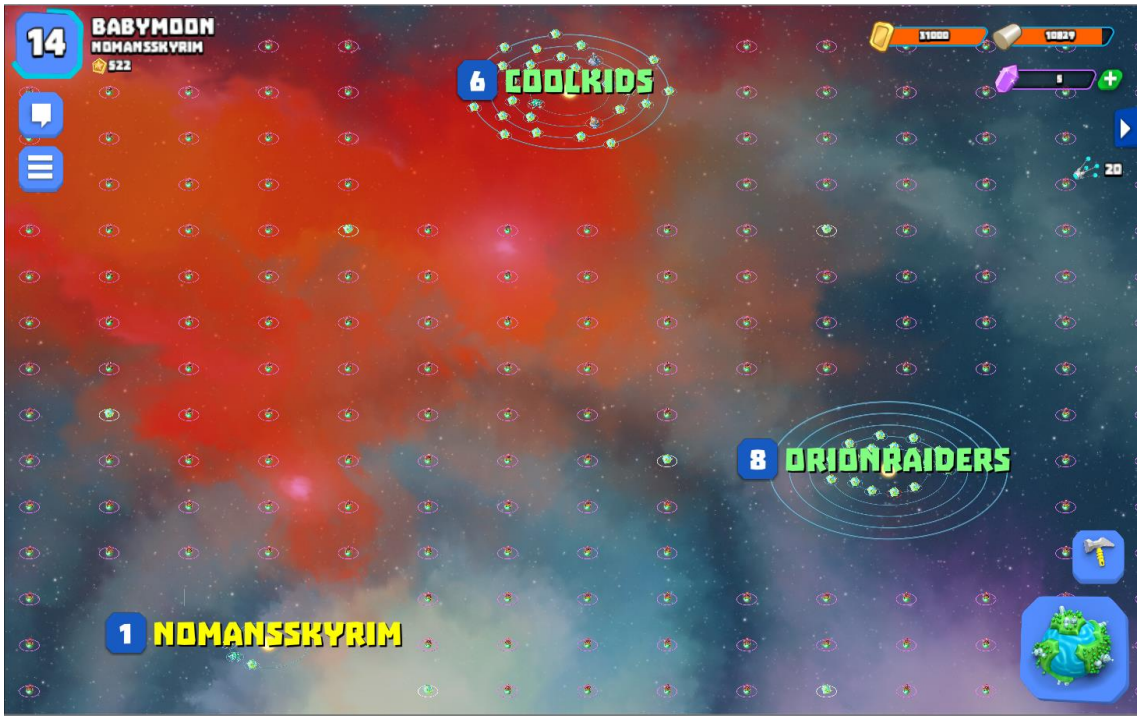


Figure 13. Initial grid pattern problem for generated normal areas

It is apparent that each normal area is capable of containing 16 planets. However, the pattern portrayed in Figure 13 is not what we want as it is very obvious that the world is computationally generated in an uninteresting way. To address this issue, the jittered grid method discussed in Section 4.2 is applied to vary the position of each planet by adding an offset to the existing position, while still keeping the planet within its own cell or slot.

```
// Set up pos
var offsetX = (float) offsetRand.NextDouble()*
              (gmConstant.GridSize_UnityUnit - gmConstant.PaddingSize_UnityUnit*2) +
              gmConstant.PaddingSize_UnityUnit;
var offsetY = (float) offsetRand.NextDouble()*
              (gmConstant.GridSize_UnityUnit - gmConstant.PaddingSize_UnityUnit*2) +
              gmConstant.PaddingSize_UnityUnit;
var relativePlanetPos = new Vector2(
    i%gmConstant.AreaSize_SlotUnit*gmConstant.GridSize_UnityUnit,
    i/gmConstant.AreaSize_SlotUnit*gmConstant.GridSize_UnityUnit);
var pos = relativePlanetPos + bottomLeft + new Vector2(offsetX, offsetY);
```

Listing 4. The exact position calculation for each planet within a normal area

As demonstrated by Listing 4, each area has a coordinate pivoted at the bottom left, and this coordinate is summed up along with the relative position of each planet within the area to form the final position. The final position is enhanced by adding two random offsets to either axes in the coordinate space, thus significantly improving the existing pattern.

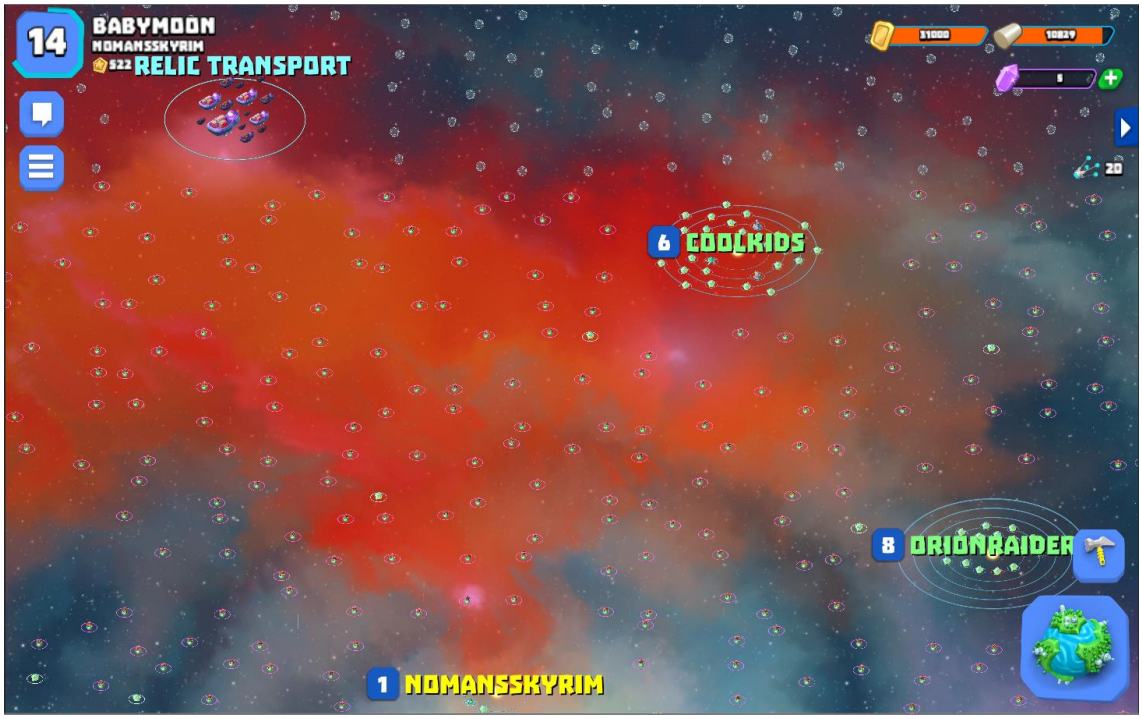


Figure 14. The improved pattern for planets on the map

While the pattern for planets has been enhanced to a greater degree, another problem remains. After discussion with designers, a conclusion was drawn that the density of the planets caused an issue with navigation, as players would find it difficult to distinct any area from the others on a much closer zoom. The overly dense distribution also overwhelms any player in the game, as they simply cannot decide the personal attachment with any neighbor nearby. To solve the problem, another deterministic trimming was needed within normal areas.

```
var random = new MathUtils.Random(areaSeed);
List<int> indicesPool = new List<int>();
List<int> planetIndices = new List<int>();
for (int i = 0; i < GalaxyMapConfig.Instance.NumberOfSlotsInArea; i++)
{
    indicesPool.Add(i);
}
for (int i = 0; i < GalaxyMapConfig.Instance.GalaxyMapConstant.PlanetsInArea; i++)
{
    int takeAway = random.Next(0, indicesPool.Count);
    planetIndices.Add(indicesPool[takeAway]);
    indicesPool.RemoveAt(takeAway);
}
return planetIndices;
```

Listing 5. Excess trimming method for normal area

Each area needs a unique trimming applied to keep a good and non-repetitive pattern across areas and therefore cannot use the same previous seed, which stems from the cluster's ID. The area's own index number should not be utilized as well since it has a limited value between 0 and 255, causing a repetition across clusters. A new seed is therefore devised from both the cluster's ID and the area's index number to cope with the pattern requirement. Listing 5 explains the method of trimming within an area by building a pool of numbers where only a few will be selected to remain in the area.

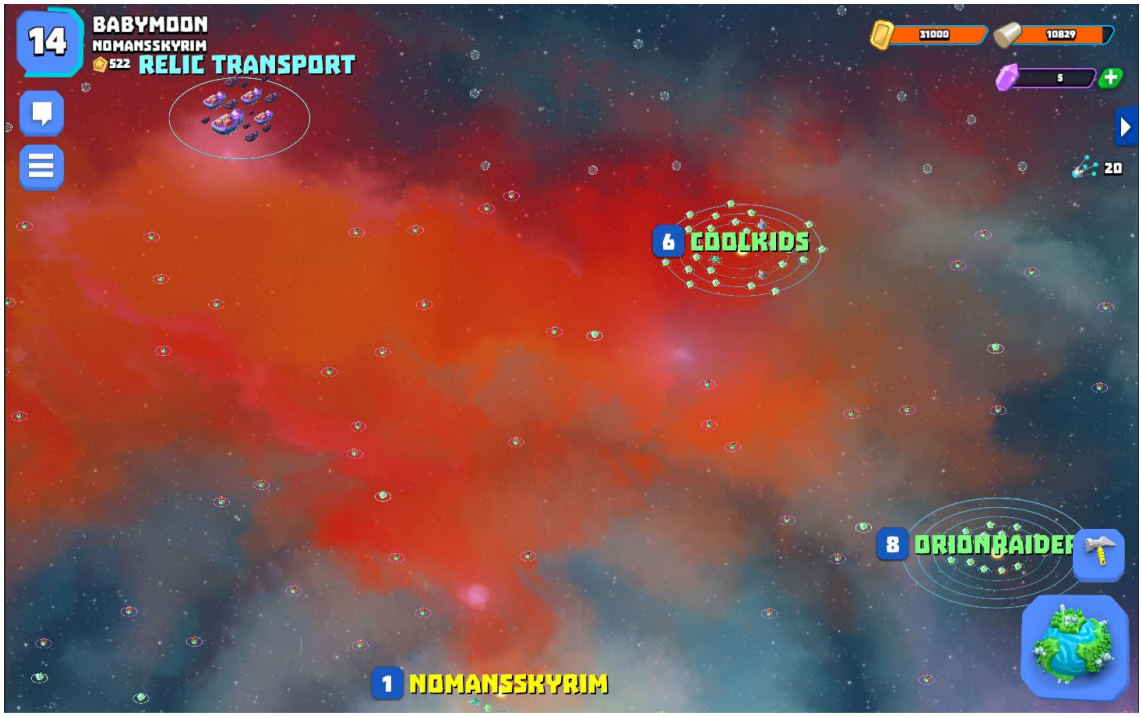


Figure 15. The final version of the generated planets pattern

Figure 15 shows the final achieved solution which solves all predictable pattern problems with both big entity areas and areas that contain planets. The procedural generation approach is refined further with the usage of random nebula backgrounds as all figures in this section have illustrated. The system is basically implemented using a pool of pre-rendered nebula images, combined together with a deterministic layering and coloring technique. The approach uses, again, the cluster's ID as the seed to generate the backgrounds bounded within the cluster region. When considered together with other big entities on the map, the generation algorithms, as a whole, achieved a reasonably good result that satisfied requirements from the designers.

5.2 Dynamic Spawning System

As concluded in Section 4.3, the map needs a single object type that is generic enough to be shaped into all other object types, thus effectively avoiding the memory fragmentation issue. A generic object has been designed as a prefab in Unity and called GalaxyObject to serve the purpose. The structure of GalaxyObject is denoted in Figure 15.

Figure 16. GalaxyObject prefab's structure

GalaxyObject is structured, as shown in Figure 16, to be capable of displaying any arbitrary static 2D sprite as well as a 2D animated sprite. The sprite can be scaled, sized or tinted as specified when the object is used. In addition to the main graphics, the object can also have an indicator and highlight the ring attached to it, and both can be freely customized according to the input parameters. Another important feature of GalaxyObject is the possibility to show the object's name and its level, as most objects on the map possess the data. Other miscellaneous features are related to the alliance system. If the object is a player who is leading an alliance, it can be marked accordingly, but if it is an alliance station, it can appear as being constructed. In a special case where the object is an alliance object, it can become a house that stores and links to other objects in the alliance.

```
public interface IGalaxyObjectModel
{
    GalaxyObjectCommon GalaxyObjectCommon { get; set; }
    AllianceStationCommon StationCommon { get; set; }
    RelicCommon RelicCommon { get; set; }
    AllianceMemberCommon AllianceMemberCommon { get; set; }

    Guid TargetId { get; set; }
    string Name { get; set; }
    byte Type { get; set; }
    byte SubType { get; set; }
    byte Level { get; set; }
    Guid ClusterId { get; set; }
    byte AreaIndex { get; set; }
    byte SlotIndex { get; set; }
    byte AllianceSlotNumber { get; set; }
}
```

Listing 6. Model data interface used by GalaxyObject prefab

To support the features mentioned above, the object needs to carry sufficient data. Listing 6 details an interface used by the GalaxyObject prefab to store the retrieved data into a model, which can then be used to trigger update on the object's view.

```
public interface IGalaxyObjectView
{
    IGalaxyObjectModel Data { get; set; }

    void Reset();

    void ActivateIndicatorGraphic(Color color);
    void DeActivateIndicatorGraphic();
    void ToggleHighlightActive(bool isActive);
    void ToggleLeaderSign(bool isActive);
    void SetOwnColor();
    void SetFriendColor();
    void SetCapturedColor();
    void SetHighlightColor(Color color);

    void UpdateStation();
    void UpdateRelic();
    void UpdateMemberCommon();
    void UpdateName();
    void UpdateType();
    void UpdateSubType();
    void UpdateLevel();
    void UpdateCoordinate();
    void UpdateTargetId();
}
```

Listing 7. GalaxyObject's view interface

Listing 7 shows how GalaxyObject's view interface is structured. This interface contains the data interface itself as well as all the necessary methods used for updating how the object looks, triggered by any changes within the data model. The interface also contains other extra methods to dynamically manipulate the attached indicator and highlight ring graphics.

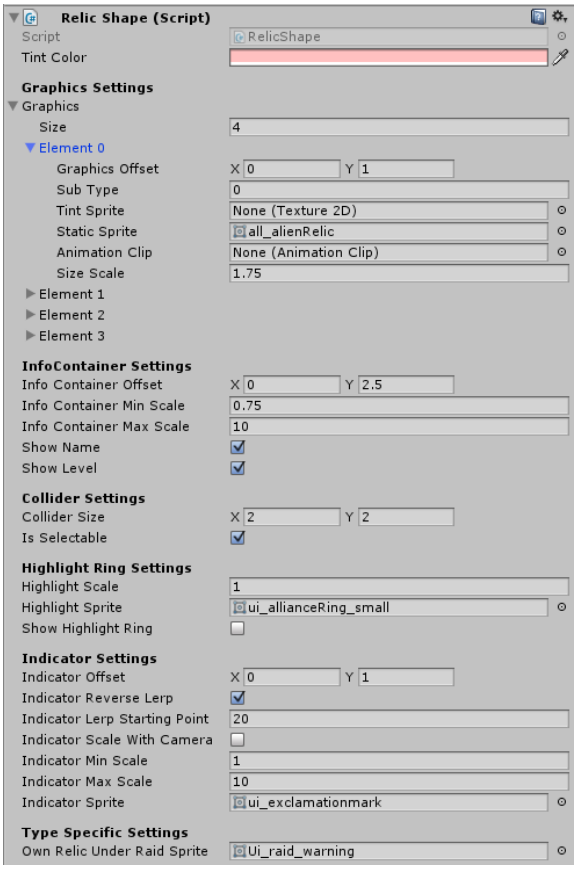
With a GalaxyObject prefab designed and prepared for usage, a dynamic spawning system is needed to populate the view, keeping the player informed of the latest data from the server. When an area comes within the view of the player, all objects within that area will be instantiated as generic objects without data. Those objects are placeholders, each having a simple text above itself, indicating the object is being loaded. In other words, its data is being fetched.

```
// Constantly fetch data to the cache as needed
private void FetchGalaxyObjects(int radius)
{
    var areasToBeRequested = GetNeedToRequestCurrentInboundAreas_Indices(radius);
    if (areasToBeRequested.Count > 0)
    {
        Events.Fire(new MapFetchAreasEvent
        {
            IsAsynchronous = true,
            InboundAreas = areasToBeRequested,
            CallBack = null
        });
    }
}

// Constantly read from the cache, initilize objects from cluster id if available
// load objects data from cache if available
private void GenerateGalaxyObjects (int radius)
{
    // Only generate objects in the current viewport (and a bit more outside)
    var inboundAreas = GetCurrentInboundAreas_Indices(radius);
    foreach (var item in inboundAreas)
    {
        AreaManager.Instance.InstantiateAreas(item.Key, item.Value, true);
    }
}
```

Listing 8. The core mechanism of Galaxy Map's dynamic spawning system

As indicated in Listing 8, the spawning system also has a fetch data method which is called as frequently as the instantiate method. However, the method used for instantiating objects involves all the areas within the view, added with some more areas padding the surrounding of the view, while the fetch method involves the same set of areas, except for the areas whose data is already stored in the cache. For the instantiated areas, they will make use of the existing data in the cache, if available, to immediately modify all objects within the area to match the data. On the other hand, if the data is not available yet, the fetching mechanism contains an internal callback, which is then triggered once the data arrives to perform update on the corresponding area. Either way, the pre-created placeholder objects within any areas will be updated according to their type, represented by the data.

```
public Color TintColor;

[Header("Graphics Settings")]
public List<GalaxyObjectGraphics> Graphics;

[Header("InfoContainer Settings")]
public Vector2 InfoContainerOffset;
public float InfoContainerMinScale = 1f;
public float InfoContainerMaxScale = 10f;
public bool ShowName;
public bool ShowLevel;

[Header("Collider Settings")]
public Vector2 ColliderSize;
public bool IsSelectable;

[Header("Highlight Ring Settings")]
public float HighlightScale = 1f;
public Sprite HighlightSprite;
public bool ShowHighlightRing;

[Header("Indicator Settings")]
public Vector2 IndicatorOffset;
public bool IndicatorReverseLerp = false;
public float IndicatorLerpStartingPoint = 3f;
public bool IndicatorScaleWithCamera = false;
public float IndicatorMinScale = 1f;
public float IndicatorMaxScale = 10f;
public Sprite IndicatorSprite;

public virtual GalaxyObjectType Type
{
    get { return GalaxyObjectType.Empty; }
}
```



Listing 9. GalaxyObjectShape's abstract class and example usage in case of Relic Shape

To update an object to reflect its data, the factory method pattern is used as an outcome of the study in Section 4.5. Listing 9 details the structure of the abstract class GalaxyObjectShape, which is derived by multiple classes, each representing an object type, namely RelicShape, AllianceStationShape, etc. Listing 9 also illustrates an example of RelicShape class, when exposed to Unity's inspector. The class instance is now a factory that can be used to process any GalaxyObject instance and corresponding data passed into it. The outcome is a final object which is shaped exactly as defined by the parameters in the RelicShape factory instance.

As areas and their objects coming into the view, instantiated and shaped into a desired objects, the next problem to solve is the cleaning up process of objects that no longer remain within the player's view.

```
public Transform Produce()
{
    Transform obj;

    if (_objectPool.Count > 0)
    {
        obj = _objectPool.Pop();
        obj.gameObject.SetActive(true);
    }
    else
    {
        obj = Instantiate(ObjectPrefab, Vector3.zero, Quaternion.identity) as Transform;
    }

    _producedObjs.Add(obj);

    return obj;
}

public void Recycle(Transform obj)
{
    if (obj == null) return;

    obj.gameObject.SetActive(false);
    obj.parent = transform;
    _objectPool.Push(obj);
    _producedObjs.Remove(obj);
}
```

Listing 10. Pooling system's core methods

The dynamic spawning system does not only call the area instantiate method constantly, but it does the same for the area recycle method. The system basically ensures that before any new area is created, the old ones must first be recycled. This guarantees the same objects will be recycled over and over again, thus avoiding the need to allocate new memory space for new objects. To facilitate such behavior, a pooling system is required. Listing 10 shows the most basic methods that any pooling system requires. Every time an object is needed, the object will be taken away from the pool if available. Otherwise a new object will be created through the traditional memory allocation method. All produced objects will be kept track of from a list. When an object is removed, it will also be removed from the list and stored back into a stack where it can be popped out later.

When the game is run on a device, it is important to keep the memory usage lower than the memory usage threshold. When the threshold is exceeded, the game will be terminated by the OS to ensure system stability. This has been the case for Last Planets where the game consistently crashes whenever the memory usage exceeds the threshold of approximately 260 MB on iPad 2, along with several other iOS devices which have

only 512 MB of RAM. The Galaxy Map is not the only system that can occupy RAM in the game. When the player switches from the map to other scenes, those scenes will also create their own objects. This will, in turn, result in a peak in the RAM usage since the game needs to sustain memory for several scenes at the same time to support instant scene switching. To avoid the peak and prevent a crash from happening, a better pooling system will be needed.

```
public IEnumerator EmptyThePool()
{
    while (_objectPool.Count > 0)
    {
        var obj = _objectPool.Pop();
        Destroy(obj.gameObject);
    }
    yield return null;
    GC.Collect();
}

public IEnumerator FillThePool()
{
    yield return StartCoroutine(WaitTillPoolIsStablized());
    if (GameStateManager.Instance.CurrentState() != GameState.GalaxyMap) yield break;

    var targetObjectPoolSize = PreWarmPoolSize - _producedObjs.Count;

    while (_objectPool.Count < targetObjectPoolSize)
    {
        var t = Instantiate(ObjectPrefab, Vector3.zero, Quaternion.identity) as Transform;
        Recycle(t);
    }
    GC.Collect();
}
```

Listing 11. Dynamic pool adjustment between scenes

Listing 11 demonstrates the implementation of additional utility methods for the pooling system. They are called at the most opportune moments to optimize the RAM usage, while still ensuring the performance on the map. When the game is switched away from the map, the pool will be forced to be emptied, after which the GC.Collect method is called to force the collection of unused memory blocks. As soon as the game state is changed back to the map, the pool will wait until the object production phase is stabilized, at which point the method can determine how many objects are needed to instantiate to fill the pool. The PreWarmPoolSize parameter is used for the pool filling strategy to define the optimal pool size needed for the optimal performance. The GC.Collect method is called again afterwards to force garbage collection of the previous scene.

5.3 Network and Cache Implementation

In order to optimize the network request usage, a cache system has been implemented and revolved around all data requests that occur within the Galaxy Map. When the game is first started, a request will be sent to the server to fetch all existing clusters in the universe, including their relative positional data.

```
public void InitGalaxyMap(ClusterCommon[] allClusters)
{
    // Initialize hashtables
    _clusterIdTable = new Dictionary<Guid, ClusterCache>();
    _clusterCoordinateTable = new Dictionary<Point2, ClusterCache>();
    _hashTable = new Dictionary<int, Guid>();

    // Insert all the clusters and mark them as requested
    foreach (var cluster in allClusters)
    {
        InsertClusterAt(cluster.ClusterId, cluster.PositionX, cluster.PositionY);
    }
}
```

Listing 12. Initialization of the map's cache

As shown in Listing 12, it can be noted that the whole map is reconstructed on the client after data is received from the server. The cache construction includes the usage of instant lookup dictionaries by storing each cluster's ID along with its coordinate and hash for a quick access to the cache as it will be referenced very frequently. The ClusterCache class of objects, which plays an important role in how data is cached for a cluster, is introduced in Listing 13.

```
public Guid ClusterId { get; private set; }
public Point2 Coordinate { get; private set; }

public Dictionary<byte, Dictionary<byte, GalaxyObjectCommon>> GalaxyObjectsDictionary;

public byte[] EmptyAreaIndices { get; }
public byte[] AllianceAreaIndices { get; }
public byte[] EventAreaIndices { get; }
public byte[] RelayAreaIndices { get; }

private readonly Dictionary<byte, AreaCacheState> _areaRequestTable = new Dictionary<byte, AreaCacheState>();
```

Listing 13. Basic structure of ClusterCache class

The ClusterCache stores and represents all the data a cluster can have. The most basic data are the cluster is own ID and its coordinate. It also caches all reserved area types by index numbers since the procedural generation process is a relatively expensive operation which should not be executed in any loop when its results are needed. The cache provides GalaxyObjectsDictionary as the most important storage on the map. It is a two-level cache, with the first level being an area cache while the second level serves as a per object cache. This allows extremely fast reference to the object data, provided that its coordinate is known. The second most important cache in this structure is the area request table, with the core AreaCacheState class being shown in Listing 14.

```
public AreaType AreaType;
public DateTime RequestDate;
public int Hash;
public bool Requested
{
    get { return DateTime.UtcNow - RequestDate < TimeToLiveTable[this.AreaType]; }
}

private static readonly Dictionary<AreaType, TimeSpan> TimeToLiveTable = new Dictionary<AreaType, TimeSpan>
{
    {AreaType.Alliance, TimeSpan.FromSeconds(Conf.Instance.NetworkConstant.AllianceAreaCacheExpiration)},
    {AreaType.Empty, TimeSpan.FromSeconds(Conf.Instance.NetworkConstant.EmptyAreaCacheExpiration)},
    {AreaType.Event, TimeSpan.FromSeconds(Conf.Instance.NetworkConstant.EventAreaCacheExpiration)},
    {AreaType.Normal, TimeSpan.FromSeconds(Conf.Instance.NetworkConstant.NormalAreaCacheExpiration)},
    {AreaType.Relay, TimeSpan.FromSeconds(Conf.Instance.NetworkConstant.RelayAreaCacheExpiration)}
};

public AreaCacheState(AreaType areaType, int hash)
{
    AreaType = areaType;
    RequestDate = DateTime.MinValue;
    Hash = hash;
}
```

Listing 14. AreaCacheState's structure

Each AreaCacheState instance caches the state of an area within a cluster. As noted before, some areas might have been requested, while others have not, this is used to determine whether or not a request should be made, thus optimizing the network resource usage. Apart from the basic area type and area hash, which are both used for fast reference, the requested state of the area is not a simple Boolean value. Instead of using a Boolean value which requires the manual work of toggling the requested state to either true or false, a request date and configuration variables are used to realize the implementation. Once an area is requested and its data is received, the status of the cache for this area can only stay valid for a limited amount of time. This was discussed briefly in the section 4.4 and concluded as the solution to solve the request interval problem.

How long the cache for a specific area type can remain valid was discussed with designers to consider the most important ones for the game. In addition to that, discussion with the backend programmer was also carried out to measure and fine-tune the cache expiration time. Priority has been given to event areas with the refresh rate as low as every 10 seconds, alliance areas come in second in the degree of importance with normal areas being the least important of all. This is due to the fact that the social gameplay is the focus where players should be encourage to engage in the most up-to-date social activities. However, the parameters have all been moved to the configuration file, which is supplied over the air as the game starts, giving the advantage of dynamic adjustment to fit the bandwidth limitation without having to update the client.

6 Results and Discussion

6.1 Results

From the technical perspective, the implementation of the Galaxy Map has achieved the goal of being usable on all iOS devices within the target market. The most low-end devices within the market range are the ones that are equipped with 512 MB of RAM. One example of this kind of device is iPad 2. Various tests have been carried out to profile the RAM usage on those devices while the player is active on the map. The RAM usage stabilizes around 220-230 MB in most cases, with the exception of being a little higher when switched to another scene that has many objects created and switched back. The game does not crash in general as RAM usage rarely peaks at more than 260 MB for the whole game. However, the game does crash occasionally after a long session, although it is quite rare. This ensures that the game can support and run on a lower-end hardware market segment, thus enlarging the user base.

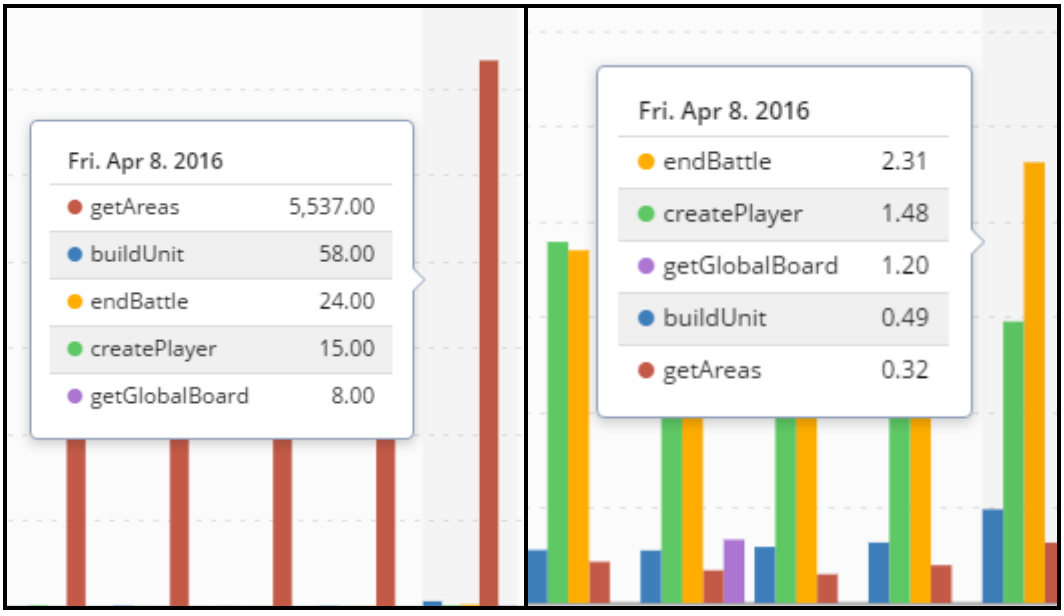


Figure 17. Network requests amount and requests' cost for several operations

In terms of network usage performance, Figure 17 is provided by the backend developer regarding the number of requests and the cost of each request during 8th April as a day of peak traffic. It is clearly shown that the number of getAreas requests is very high compared to other requests. This is, however, an acceptable number, since many players seem relatively active when exploring the map. The cost of the getAreas request is also the lowest as it is well optimized on the backend, making it suitable for a mass number of requests. There is also no noticeable amount of delay while the map is being explored by testers. It can be concluded that the network performance delivered on the map is fairly acceptable.

The Galaxy Map also managed to achieve the goal for a procedural generation system in a multiplayer game. All players in the game can share the same experience, as the appearance of the map is persistent among everyone in the game. Players can also develop a sense of attachment to the environment that they are spawn into. This is achieved through the usage of unique pattern generation for various landmarks or object types across the map, while at the same time ensuring that players are not overwhelmed by an environment that is exceedingly crowded.

6.2 Discussion

With the crash issue still occurs in rare events, which is linked to the peak RAM usage, initial investigations were carried out to find the cause. The result strongly hinted on the total RAM usage by all scenes in the game being exceedingly high. The most probable reason for this phenomenon is the lack of pooling usage for building objects in other scenes, as they can remain dormant in the memory with no purpose after the game state is switched to the map. Usage of the pooling technique and a single shared object pool using a generic object type can be extended to the whole game to improve the performance and memory usage. However, the move to use such a system will be costly and should be carefully considered.

One major concern regarding the procedural generation method is the fact that it is not as competitive as the traditional hand-made design provided by artists, as many recent procedurally generated open world games have been proven to be repetitive and dull. The Witcher 3 is an example of a very successful RPG video game that is designed with an open world approach. However, the game does not use any procedural generation methods. The result of the in-game world is merely the product of collaboration and hard work of all the people who participate in the development. (Klepek 2015.)

Nevertheless, the Galaxy Map in Last Planets, while relies mostly on procedural generation, does require the content creation process from players through the alliance system. This can add more uniqueness to the game as those landmarks are player-generated. Moreover, while this report does introduce many techniques and methods that can be used to achieve a procedural generated environment, it should be noted that those are just basic solutions. As pointed out in Section 3.3, games such as Minecraft and Dwarf Fortress have attained very good successes through the usage of procedural generation. As a result, I believe deeper and more extensive usage of those techniques can yield a much better variety and realism to enhance the experience for the Galaxy Map in the future.

7 Conclusions

With the goals of designing the Galaxy Map to be a procedurally generated virtual platform and to be a core feature which facilitates virtual interaction between players in Last Planets, the project reached its goals after the development process of more than half a year.

The new implementation has managed to replace the previous placeholder implementation to support a scalable universe. Thus, the design goal of making the Galaxy Map an interactive platform between thousands of players has been reached. The map has achieved not only the target of being a functional virtual world, but also succeeded in using various procedural generation techniques to create an unlimitedly expandable universe without going through a manual creation process.

Both sophisticated and simple methods were combined throughout the study to effectively generate an interesting universe that is unique for any player who is spawn into a specific location on the map. As for the performance, the map performs rather well both in terms of resource usage on the client and on the server. With the possibility of allowing the exploration of a vast dynamic networked virtual world, the achievement in RAM usage on low-end client devices marked a success in ensuring wide target audiences. However, RAM usage can still be optimized further to prevent occasional crashes in certain cases.

The knowledge explored and used in this study is still rather limited and basic, but can serve as an ideal starting point reference for developers who want to develop similarly designed games. Procedural generation in a video game is, however, a very broad topic with many challenging difficulties from both technical and design perspectives. Even with just a basic implementation of the Galaxy Map without using any advanced procedural generation solutions gathered from other games, the scope and length of the study is already near its limit. For that reason, it is very difficult to include all the details related to how the Galaxy Map was implemented and thus anything that did not focus on the subject of the study had to be left out. Further research into this area can be done to modify the current implementation and improve the unique appearance of the map. However, it will be a subject of future consideration after the game is launched.

References

Bridson Robert. Fast Poisson disk sampling in arbitrary dimensions. In: ACM SIGGRAPH '07 Sketches & Applications. ACM Press. Article No. 22; 2007

Champandard Alex. Beyond Terrain Generation: Bringing Worlds in DWARF FORTRESS to Life [online]. AiGameDev; 23 April 2012. URL: http://aigamedev.com/open/teaser/living-worlds-dwarf-fortress/. Accessed 25 April 2016.

Dickinson Chris. Unity 5 Game Optimization. Birmingham: Packt Publishing; 2015.

Freeman Eric. Head First Design Patterns. 1st Edition. O'Reilly Media; 2004.

Fiedler Glenn. UDP vs. TCP [online]. Gaffer on Games; 2008. URL: http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/. Accessed 25 April 2016.

Fingas Jon. Here's how 'Minecraft' creates its gigantic worlds [online]. Engadget; 4 March 2015. URL: http://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/. Accessed 25 April 2016.

Gregory Jason. Game Engine Architecture, Second Edition. A K Peters/CRC Press; 2014.

Klepek Patrick. How The Witcher 3's Developers Ensured Their Open World Didn't Suck [online]. Kotaku; 6 October 2015. URL: http://kotaku.com/how-the-witcher-3s-developers-ensured-their-open-world-1735034176. Accessed 25 April 2016.

Khatchadourian Raffi. World Without End [online]. The New Yorker; 18 May 2015. URL: http://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian. Accessed 25 April 2016.

Lernö Christoffer. Game servers: UDP vs TCP [online]. 1024 Monkeys; 1 April 2014. URL: https://1024monkeys.wordpress.com/2014/04/01/game-servers-udp-vs-tcp/. Accessed 25 April 2016.

Minecraft Wiki. Coordinates [online]. Minecraft Wiki; 2016. URL: http://minecraft.gamepedia.com/Coordinates. Accessed 25 April 2016.

My Abandonware. Beneath Apple Manor [online]. My Abandonware; 2016. URL: http://www.myabandonware.com/game/beneath-apple-manor-256. Accessed 25 April 2016.

MSDN. Guid Structure (System) [online] MSDN; 2016. URL:https://msdn.microsoft.com/en-us/library/system.guid(v=vs.110).aspx. Accessed 25 April 2016.

Maher Jimmy. Akalabeth [online]. The Digital Antiquarian; 18 December 2011. URL: http://www.filfre.net/2011/12/akalabeth/. Accessed 25 April 2016.

Newman, E.L.; Caplan, J.B.; Kirschen, M.P.; Korolev, I.O.; Sekuler, R.; Kahana, M.J.; et al. Learning Your Way Around Town: How Virtual Taxicab Drivers Learn to Use Both Layout and Landmark Information. In: Cognition Volume 104 (2); 2007. p. 231–253.

Susman Gerald and Evered Roger. An Assessment of the Scientific Merits of Action Research. In: Administrative Science Quarterly. Volume 23 (4); 1978. p. 582-603.

Tulleken Herman. Poisson Disk Sampling [online]. Dev.Mag; 3 May 2009. URL: http://devmag.org.za/2009/05/03/poisson-disk-sampling/. Accessed 25 April 2016.

Venkatesh CM. Caching To Scale Web Applications [online]. Venkatesh CM; 17 May 2014. URL: http://venkateshcm.com/2014/05/Caching-To-Scale-Web-Applications/. Accessed 25 April 2016.

Wolf Mark and Perron Bernard. The Video Game Theory Reader 2nd Edition. Abingdon: Routledge; 2008. p. 339-340.